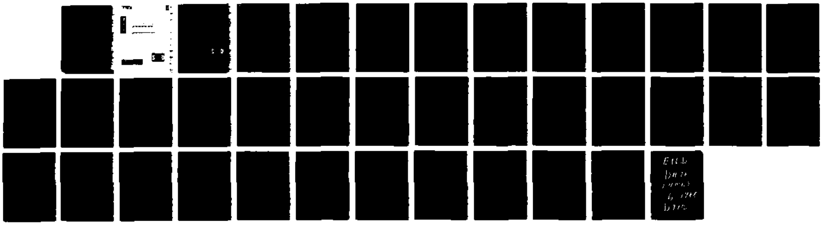
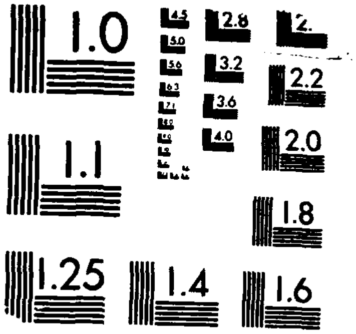
AD-A192 544 A TECHNIQUE FOR CONSTRUCTING HIGHLY AVAILABLE SERVICES (U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE R LADIN ET AL. FEB 88 MIT/LCS/TR-409 1/1

UNCLASSIFIED F/G 12/8 NL




1.0
4.5
5.0
5.6
6.3
2.8
2.
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

4

MIT/LCS/TR-409

A TECHNIQUE FOR CONSTRUCTING HIGHLY AVAILABLE SERVICES

Rivka Ladin
Barbara Liskov
Luiba Shrira

February ~~January~~ 1988

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-409	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125

6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science	6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS *(City, State, and ZIP Code)* 545 Technology Square Cambridge, MA 02139		7b. ADDRESS *(City, State, and ZIP Code)* Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION DARPA/DOD	8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Blvd. Arlington, VA 22217		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
A TECHNIQUE FOR CONSTRUCTING HIGHLY-AVAILABLE SERVICES

12. PERSONAL AUTHOR(S)
Ladin, Rivka; Liskov, Barbara; Shrira, Liuba

13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM ________ TO ________	14. DATE OF REPORT *(Year, Month, Day)* February 1988	15 PAGE COUNT 30

16. SUPPLEMENTARY NOTATION

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
distributed systems, algorithms, reliablity, availability, data replication

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This paper describes a general method for constructing a highly-available service for use in a distributed system. It gives a specific implementation of the method and proves the implementation correct. The service consists of replicas that reside at several different locations in a network. It presents its clients with a consistent view of its state, but the view may contain old information. Clients can indicate how recent the information must be. The method can be used in applications satisfying certain semantic constraints. For applications that can use it, the method performs better than other replication techniques.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b TELEPHONE *(Include Area Code)* (617) 253-5894	22c. OFFICE SYMBOL

83 APR edition may be used until exhausted.
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Unclassified

A TECHNIQUE FOR CONSTRUCTING HIGHLY-AVAILABLE SERVICES

by

Rivka Ladin
Barbara Liskov
Liuba Shrira

December 1987

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125, by the National Science Foundation under grant DCR-8503662 and by the HTI Postdoctoral Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Table of Contents

1 Introduction 1
2 Orphan Detection 2
3 Implementing the Map Service 6
3.1 Optimizations 8
4 The General Replication Technique 11
5 Performance 13
6 Conclusions 15
References 17
I. Appendix 19
I.1 The Model 19
I.2 Implementation 19
I.2.1 Execution of the Service 20
I.2.2 Correctness of the Service 23
I.3 Client Interface 28

A Technique for Constructing Highly-Available Services

Rivka Ladin
Parbara Liskov
Liuba Shrira

MIT Laboratory for Computer Science
Cambridge, Ma. 02139

Abstract

This paper describes a general method for constructing a highly-available service for use in a distributed system. It gives a specific implementation of the method and proves the implementation correct. The service consists of replicas that reside at several different locations in a network. It presents its clients with a consistent view of its state, but the view may contain old information. Clients can indicate how recent the information must be. The method can be used in applications satisfying certain semantic constraints. For applications that can use it, the method performs better than other replication techniques.

Keywords: distributed systems, algorithms, reliability, availability, data replication

1 Introduction

This paper describes a general method for constructing a highly-available service for use in a distributed system. It gives a specific implementation of the method and proves the implementation correct. The service presents its clients with a consistent view of its state, but the view may contain old information. Clients can indicate how recent the information must be. The method was invented as a way of optimizing the orphan detection strategy developed for the Argus language and system [10], ~~[11]~~, but appears to be applicable to a wide range of applications, including garbage collection of objects in a distributed heap [12], locating movable objects in a distributed system [8], and deletion of unused versions in a hybrid concurrency control scheme [19]. It requires that applications satisfy certain semantic constraints. For such applications, the method performs better than other replication schemes.

The method is intended to be used in an environment in which individual computers, or *nodes*, are connected by a communications network. Both the nodes and the network may fail; the method tolerates these failures. The nodes are failstop processors [17]; we assume they can crash, but Byzantine failures are not expected. We assume that nodes do eventually recover from crashes, and that each node has access to a stable storage device that (with very high probability) preserves the information entrusted to it [9]. After a crash, a node can recover the portion of its state that was written to its stable storage device before the crash.

The network connecting the nodes can have an arbitrary topology. For example, it might consist of a number of local area nets connected via gateways to a long-haul network. Again we rule out Byzantine failures, but otherwise the network can behave arbitrarily badly. For example, it can partition. Therefore messages can be lost, delayed, duplicated, and delivered out of order.

We are concerned here with a distributed implementation of a logically centralized service. Information stored by the service is replicated at different nodes of the network. Reasons for implementing a service in this way are high availability and low response time. However, in spite of its distributed implementation, the service is logically centralized in the sense that it appears as a single entity to clients. Clients interact with the service by calling its operations. Operation calls are atomic: they are mutually indivisible, and a call either happens completely or not at all.

When data of a service is replicated at several nodes, the information at the different replicas may not be identical. Clients can observe the information at the replicas only by calling service operations, and the implementation of the operations can hide the inconsistencies. One way to implement operations is to use a voting scheme as described in [6]. In such a scheme, the nodes visited by operations that modify the service state must intersect those visited by operations that either read or modify the state. There is some freedom in choosing how to implement the operations; for example, if there are three nodes, information could be written to three nodes and read from only one, or information could be both read and written at two nodes. The former choice enhances the availability and response time of reading; availability of writing is poor since all three nodes have to be up and accessible, although this problem can be mitigated to some extent by using the technique described in [4], [3]. Any choice made for reading and writing results in at least some operations having reduced availability and requiring more message passing than is needed if the service is not replicated.

An attractive alternative to voting schemes is to have both reading and writing take place at just one replica; any convenient replica can be used. Replicas communicate new information among themselves in background mode by exchanging "gossip" messages. Gossip schemes in which a replica resides at every client node that uses the service are described in [5], [20]. As long as things are running well, information will propagate quickly, so the amount of time replicas contain different information will be small. If there are crashes and partitions, however, differences can persist for long periods of time.

Not all applications can use a gossip scheme. For example, if withdrawals from a bank account could be made independently at different replicas there would be no way of preventing overdrafts. Other applications are more forgiving. For example, in a mail system (e.g., [2]), a request to read mail need not produce all messages that have been sent; a promise of timely delivery is sufficient. However, there may be limits on how "old" the information used actually is. For example, it may be unacceptable for a user to see a message that he deleted the last time he logged in. The above gossip schemes have no way of ensuring that information seen by clients is "recent enough."

This paper describes a gossip scheme that exploits the semantics of the applications to provide high availability. The scheme makes use of just a few replicas. Also, clients have a way of ensuring that the information returned is "recent enough."

The remainder of this paper presents our approach. We begin by describing a particular service to be implemented by the technique. Then we describe how that service is implemented; a proof that our implementation works is given in the appendix. Next we generalize our technique and characterize what applications can use the technique and how different requirements of applications affect the implementation method. The following section compares the performance of our method with some other replication methods. We conclude with a summary and discussion of future work.

2 Orphan Detection

To make our presentation concrete and to motivate our approach, we will describe how the replication method works in the context of the particular application for which it was invented. The application is orphan detection in the Argus system. Only part of our orphan detection strategy is described below; the full method is discussed in [13].

Computations in Argus run as *atomic transactions*, or *actions* for short. An action is an *orphan* if it is active but an event has occurred that will prevent it from committing. One kind of event that causes orphans is a node crash. If an action ran at a node, and then the node crashed while the action continued to run elsewhere, the action cannot commit because any modifications it made at the node, and also any locks that were being held there on its behalf, are lost in the crash.

The logical nodes of the Argus system are called *guardians*. Each guardian resides on a single physical node and communicates with other guardians by means of remote procedure calls. An action starts at one guardian and spreads to others by means of remote calls. When an action commits, the guardians visited by the action participate in the two-phase commit protocol [7] to ensure that the action either commits everywhere, or aborts everywhere if committing is impossible.

We detect crash orphans by keeping track of crashes of guardians. Each guardian maintains a local *crash count* and a *map*. The crash count is an integer that is incremented after every crash; it counts the number of times its guardian has crashed since it was created. The map stores a crash count for each guardian. (Information is not stored explicitly in the map for guardians that have nct yet been created; all such guardians are mapped implicitly to a crash count of zero.) After a guardian recovers from a crash, it increments its crash count and stores the new value in stable storage. It also updates its map to reflect its new crash count.

We also maintain a *dependency list* for each action; the list records each guardian visited by the action and its crash count at the time of the visit. The list is sent with the action as it moves from guardian to guardian. The dependency list and map together can be used to detect orphans. For example, suppose that action A visited guardian G when G's crash count was 3. If later A visits guardian H, and H's map contains a larger crash count than 3 for G, we can deduce at H that A is an orphan. When H discovers this it causes A to abort.

Our orphan detection algorithm requires that when a guardian G processes a message from guardian H, G's map must contain information at least as recent as that in H's map. (The reason for this requirement is discussed in [18], [13].) We satisfy this condition by sending H's map in the message; G can then merge any new information into its map to satisfy the condition.

Thus, every message M contains the map of its sending guardian. In addition, each message sent on behalf of a remote call (this message is either a call message or a reply message) contains the dependency list of its action.

Orphans are recognized as part of the processing of incoming messages. When a message M arrives at guardian G, it is processed as follows:

1. **The dependency list of each action A that is running at G is compared with M's map. If some guardian in the A's dependency list has a higher crash count in M's map than in the list, A is an orphan.**

2. **The maps are merged by retaining the larger of the two crash counts for each guardian, and the merged map replaces G's map.**

3. **If the incoming message is a call or reply message, then M's dependency list is compared with G's map, and if any guardian in common has a larger crash count in G's map, the calling or replying action is an orphan, and the message is discarded.**

In addition, a guardian's map is written to stable storage whenever the guardian prepares during 2-phase commit, so that the guardian can recognize orphans properly after it recovers from a crash.

The problem with the above algorithm is that the map can be very large since it must contain an entry for every guardian that ever existed. Not ^nly must this information flow in messages, thus consuming bandwidth, but it must be processed whenever a message is received, which can be costly. The central service technique was invented to solve these problems. The idea is to store the map at a central *map service*, and to associate with each distinct map value a unique timestamp. The timestamp is sent in messages instead of the map. Timestamps are ordered, and later timestamps correspond to maps containing more recent information.

The map service provides three operations:

1. The *enter* operation

   enter (guardian-id, integer) returns (timestamp)

   is used to advance a guardian's crash count; it is called when a guardian is first created, and each time a guardian recovers from a crash. It takes the guardian's identifier (the *guardian-id*) and its current crash count as arguments. The service installs the crash count in its map, increments its timestamp, and stores both on stable storage. Then it returns its timestamp.

2. The *refresh* operation

   refresh (timestamp) returns (map, timestamp)

   is called whenever a guardian needs more up-to-date information. It takes a timestamp as an argument; it returns a map and a timestamp. The returned timestamp is guaranteed to be greater than or equal to the argument timestamp. (The reason for the timestamp argument will be explained shortly.)

3. The *delete* operation

   delete (guardian-id) returns (timestamp)

   is called to notify the service that a guardian has been destroyed. It takes the guardian-id as an argument. It records information about the destruction in its map, increments its timestamp, stores both on stable storage, and then returns the timestamp.

Each message M has a timestamp, M.*ts*; this is the timestamp of its sending guardian. The receiving guardian G also has a timestamp, G.*ts*. When G receives a message, the following processing takes place:

1. If M.*ts* = G.*ts*, then no orphan detection need be done since both the sender and receiver have the same map.

2. If M.*ts* < G.*ts*, there is no danger of any actions running at G being recognized as orphans due to information in M because all crash counts in G's map are equal to or greater than those in the map of the sender of M. However, the information in G's map is used to detect whether a call or reply message is an orphan.

3. Otherwise, G calls the *refresh* operation to obtain an up-to-date map and timestamp. It uses the new information to detect and destroy any of its running actions that may be orphans. Then it proceeds as in step 1 or 2 above.

G no longer needs to store its map in stable storage, since the service is responsible for this. However. it does need to keep its current crash count in stable storage.

Orphans can be detected more rapidly using the service than in the original algorithm. The reason is that the map returned by the *refresh* operation may contain more recent information than what was available by merging a guardian's map with the map in an incoming message. In essence, our algorithm relies on propagation of information as messages flow around the system. Since everyone communicates with the service, it provides a communication path that allows especially fast propagation. Fast propagation is good since it allows orphans to be detected sooner.

Using a map service is efficient if guardians usually need not communicate with the service before processing a message. Communication at such a time can be avoided for two reasons. First, since crashes are rare, the map changes infrequently. Therefore, the system is likely to have long periods of time in which all guardians store the same map and timestamp. In addition, guardians can communicate with the service periodically in background mode to keep their map and timestamp up to date and can detect local orphans when responses from the service arrive. Therefore little work need be done to detect orphans when messages from other guardians arrive.

However, if the service resides on a single node, inaccessibility of that node because of a crash or partition would be a problem. Guardians could not recover from crashes, and incoming messages might not be able to be processed. Instead the service must be replicated so that it will be highly available. In the next section we describe how such a service can be implemented.

In our implementation, *enters* and *deletes* are executed at just one replica. When a replica carries out a *refresh* operation, it might have out-of-date information in its map. For example, it might not know about some *enter* that is recorded in either the sending or receiving guardian's current map. The orphan algorithm, however, requires that the new map at the receiving guardian record all these events. Therefore, we need a way of ensuring that *refresh* returns information that is recent enough.

We solve by this problem by passing a timestamp as an argument to *refresh*. Our timestamps are partially ordered as explained further below. Larger timestamps are associated with more recent states, i.e., those mapping guardians to larger timestamps. Timestamps can be merged; merging t and s produces a timestamp r that is greater than or equal to both t and s.

To ensure that *refresh* returns recent information, the receiving guardian (in step (3) above) merges its timestamp with that of the message, and sends the merged timestamp as an argument of *refresh*. The timestamp returned by *refresh* is guaranteed to be greater than or equal to this timestamp, and therefore the map returned by *refresh* has information at least as recent as the guardian's old map and the sender's map.

3 Implementing the Map Service

This section describes how to implement the map service to provide high availability. In what follows we describe the processing and the stored information in a general way, without considering possible optimizations; some optimizations are discussed in Section 3.1. Before discussing the implementation, however, we give a more precise specification of the map service. A formal specification is given in the appendix, which also contains a proof that our implementation works correctly.

Recall that the service provides three operations:

enter (guardian-id, integer) returns (timestamp)
delete (guardian-id) returns (timestamp)
refresh (timestamp) returns (map, timestamp)

A client receives timestamps as results of executing *enters* and *deletes*, but is able to observe the state only by executing the *refresh* operation. Therefore, the specification is expressed in terms of what can be observed by calls of *refresh*:

1. If a *refresh* is after a *delete* (i.e., its argument timestamp is greater than or equal to that returned by the *delete*), it will not observe an association for the deleted guardian.

2. If a *refresh* is after an *enter*, it will observe the <guardian-id, crash-count> pair unless the *enter* has been superseded either by another *enter* for that guardian with a later crash count, or by a *delete* for that guardian.

3. The timestamp returned by a *refresh* is greater than or equal to the argument timestamp, and every <guardian-id, crash-count> pair in the returned map was added by an *enter* operation.

The service is implemented by a small number of replicas (e.g., 3 to 7). We assume in this paper that there are a fixed number of replicas residing at fixed locations. Therefore both clients and replicas know how to locate replicas.

When a service operation is called by a client, code running at the client sends an appropriate message to a replica. There is a message type corresponding to each operation. The replica responds by sending back a reply message. If the response is slow, the code at the client may send the message to a different replica, or the client might send messages to all replicas in parallel. Therefore a single operation call can result in messages being sent to several different replicas. As mentioned above, we assume messages can be lost, delayed, duplicated, and delivered out of order.

Each replica maintains a state and the timestamp of that state. The state is simply a map that associates guardian-ids with either crash counts or a special value representing a deletion. Entries are stored in this map only for guardian-ids that map to nonzero crash counts; thus, if there is no entry for a guardian-id, that guardian's crash count is zero. Initially the map at each replica contains no entries, and the timestamp at each replica is zero. Replicas respond to messages sent by operations by either looking up information in the state, or by generating a new local state and timestamp. They communicate with one another periodically by exchanging gossip messages.

The main problem that must be solved to make this work efficiently is the generation of timestamps. Replicas must be able to generate new timestamps independently, or our implementation will be dependent on a timestamp service posing exactly the same problems we are trying to solve. Furthermore, we must generate timestamps in such a way that later timestamps are associated with states containing more recent information.

We solve this problem by using *multipart timestamps*, where there is one part for each replica. Thus if there are n replicas, a timestamp t is

$$t = <t_1, ..., t_n>$$

where each part is a nonnegative integer. The initial (zero) timestamp contains a zero in each part. Since there will typically be a small number of replicas, using such a timestamp is practical. (A similar technique is described in [16].)

A replica generates a new timestamp by incrementing its part of its timestamp by one, while leaving all other parts unchanged. Since each part can be advanced by only a single replica, we guarantee that the resulting timestamps are unique. Some timestamps can be compared: For two timestamps t and s, t is less than or equal to s provided t_i is less than or equal to s_i for each part i of the timestamp. Other timestamps are incomparable. Two timestamps t and s are merged by retaining the larger value for each part; as required, the result of the merge is greater than or equal to both t and s.

The messages are processed at the replicas as follows: If a replica receives an *enter*(g, x) message, it looks in its state to see if there is an association for g. If there is no association and x is greater than zero, or if g is mapped to a value less than x, the replica associates x with g in its local state, advances its timestamp, and returns the new timestamp in a reply message. Otherwise it simply returns its current timestamp.

When a replica receives a *delete*(g) message, it looks up g in its local state. If g is already marked as deleted, it returns its current timestamp; otherwise it marks g as deleted in the local state, advances its timestamp and returns the new timestamp.

A *refresh*(t) message causes the replica to compare t with its timestamp, t_R. If t is not less than or equal to t_R, the replica needs more information. Either it waits for gossip messages from the other replicas or it sends a query to another replica to elicit the information. Communication between replicas is discussed below. As soon as $t \leq t_R$, it returns its map and t_R.

Periodically, a replica sends a gossip message containing its timestamp and its state to other replicas. When a replica receives a gossip message, it proceeds as follows: If the timestamp in the message is less than or equal to its timestamp, it discards the message since it is old. Otherwise it merges the timestamp of the message with its timestamp. The merged timestamp becomes the new timestamp of the replica. Then it merges the state of the message with its own by retaining the larger value for each association that is present in both states, and by retaining all associations that are present in only one state. The special "delete" value is treated as larger than all integers in doing the merge.

As discussed further in Section 5, it may not be necessary for replicas to store their maps in stable storage. In this case, a replica need not write to stable storage every time it processes an update. A replica's part of its timestamp must be monotonic. It could be obtained by reading the replica's clock if the clock were stable. Alternatively, the replica's timestamp part can be written to stable storage only every n increments. After a crash, the value read from stable storage would be advanced by n and written back.

3.1 Optimizations

There are a number of optimizations that can be done to make the map service run more efficiently. For example, the size of gossip messages can be reduced by sending only recent changes; see [20] for a discussion of such an optimization. The amount of information returned by *refresh* could be reduced by returning just that portion of the map not known to the caller (in this case, the caller's current timestamp would be needed as an extra argument). Also if *enter* returned the map associated with the returned timestamp, the client could avoid a call of *refresh*.

An important optimization is to discard information about deleted guardians. The implementation described above requires that such information be retained forever. This section describes such an optimization, which is based on two assumptions:

1. Clients never enter a new crash count for a guardian after it has been deleted.

2. Clocks never run backwards.

The reason for these assumptions is explained further below.

If we discard information about deleted guardian-ids, an association that has not yet been entered would be identical to one that has been deleted. Therefore, when we perform an *enter* operation, we would not know whether to associate the guardian-id with the crash count or not. Note, however, that we would have no problem if clients never enter a new crash count for a guardian after it has been deleted. This constraint holds for the map service as used for orphan detection because guardian-ids are never reused in our system. Therefore we will make this assumption about clients.

Even if clients never call *enter* after deleting an entry, there is still a problem with late messages. If an *enter* message is delivered late, it may arrive after the deleted entry has been discarded. To solve this problem, we have each message contain the time τ at which it was sent and we impose an upper bound δ on message delay. τ is the time of the local clock at the sending node; it should not be confused with the timestamps discussed above.

When a message arrives we compare its time τ with the local clock time τ_R. If

$$\tau + \delta < \tau_R$$

the message is discarded. To handle late *enter* messages that are not discarded but arrive after the *delete* message, we retain information about a deletion at least δ more than the time in the *delete* message. We need the assumption that clocks are monotonic here because we assume that once a message is found to be late at a particular replica, all earlier messages of that client will always be late at that replica.

Waiting δ is sufficient to avoid problems with late messages from clients. It does not prevent problems with gossip messages, however. Suppose replica R waited this long and then discarded an association for guardian-id g. Then it sent a gossip message to replica S, which still had an association stored for g. There would be no way at S to decide whether the association for g should be retained or not.

To solve this problem, we proceed as follows. We continue to store information about deleted entries in the state. However, in such an entry, e, we store two additional pieces of information: *e.time*, the time of the delete message, and *e.ts*, the multipart timestamp generated when the *delete* message was processed. In addition, we maintain a replica table, *ts-table*, containing a multipart timestamp for each replica. When a gossip message is processed, the timestamp in the message is stored in *ts-table* in the entry for the replica that sent the message. Note that the real timestamp of the sending replica must be at least as large as the one stored for it in *ts-table*.

It is safe to remove deleted entry *e* when:

1. its time *e.time* + δ is less than the time of the local clock of the replica, and

2. its timestamp *e.ts* is less than or equal to all the timestamps stored in the replica's *ts-table*.

The first condition takes care of late *enter* messages. The second guarantees that the entry is retained until we can be sure that a state mapping the guardian-id to a normal value will never be received in a gossip message. If *e.ts* is less than or equal to all timestamps stored in *ts-table*, it is less than or equal to the current timestamp of each replica. Therefore each replica has heard about deletion *e*. Any new gossip messages generated by a replica will contain either information about the deletion or no information for the guardian-id at all. Old, delayed gossip messages might contain bad information, but they will be discarded, as mentioned above, because they contain a timestamp less than that of the receiving replica.

One final point concerns duplicate *delete* messages processed at different replicas. In this case, the states at the replicas will contain slightly different information for the deleted guardian-id. A simple approach is for a replica to merge the information in the gossip message with its information. In this case, an entry for a deleted guardian will contain a set of <time, timestamp> pairs. Each pair in the set can be removed when it satisfies the criteria discussed above; the entry can be removed when the set is empty. There are various ways of optimizing this, however. In fact, it is safe to remove the entry as soon as there is a pair in the set satisfying the time criterion and another (possibly different) pair satisfying the timestamp criterion. Using the earliest time works since even the earliest *delete* message is later than the latest *enter* message. Using the first timestamp to satisfy the criterion works because every replica must know at least about that particular delete. Thus, we could remove the entry as soon as just one of its pairs satisfies both criteria, and, in fact, we can store just one pair in the entry. Since any pair will do, a replica can, e.g., keep the pair it has and ignore the one in the gossip message.

For the system to run efficiently, it is sufficient that clocks of replica and client nodes be loosely synchronized so that the skew of the local clocks is bounded by some ϵ. The delay δ would be defined to accommodate both the anticipated network delay and the clock skew. If clocks are not synchronized then certain suboptimal situations can arise. If a client's clock is slow, then its messages may be discarded even though it just sent them. Also, if a replica's clock is slow, it may retain information about deletes for a long time.

In reality clocks drift by very little, and the value of ϵ can be large, so that clocks may naturally be loosely synchronized. If synchronization is needed, there are various algorithms that can be used [14], [15]. The need for synchronization can be determined by the nodes themselves based on the messages they exchange: If a node receives a message whose time is earlier than its local time + σ - ϵ, where σ is the actual expected message delay, then either its clock is slow or the sender's clock is fast, so a resynchronization is needed.

Note that the correctness of our algorithm does not depend on clocks being synchronized. As mentioned earlier, we do require that clocks never run backwards, so if a clock is found to be fast, its node must stop processing messages until the new real time is at least as great as the value of the node's clock before the synchronization.

The appendix contains a proof that our method, including the deletion optimization, works.

4 The General Replication Technique

In the preceding section we discussed how to implement a central service for a particular application. In this section we discuss how the method can be applied in general and characterize the set of applications that can use the service.

Any replication method provides two categories of operations: *Update* operations modify (but do not read) the state stored in the replicas, while *query* operations access the stored information. For example, the map service provides two update operations (*enter* and *delete*) and one query operation (*refresh*). (Sometimes an operation is both an update and a query.) Our method imposes constraints on updates that make it possible for them to be performed at a single replica; we require that updates be *idempotent* and *commutative*. We also require that queries be able to make use of old information. The replication technique is suitable for any application whose operations satisfy these constraints.

Idempotency means that it does not matter how many times an operation is executed because the effect is the same as if it were executed exactly once. Updates must be idempotent because we perform each update multiple times, either in response to multiple update messages sent by the client, or simply in the process of merging states. (When new information from one state is merged into another, this is equivalent to performing the update that introduced the new information.) Idempotent updates are easy to achieve; at worst each update must take a unique identifier as an argument.

Commutativity means that the effect of executing a group of updates is the same no matter what the order of execution. Updates must be commutative because different replicas may execute them in different orders. Map service updates satisfy this requirement as follows. Updates involving different guardians are independent and therefore their execution order does not matter. Order does not matter for updates for the same guardian because a larger crash count, or a delete, always prevails. For example, if *enter*(*g*, 6) happens to be executed before *enter*(*g*, 3), the effect (once both have been performed) is the same as it would be if the opposite order occurred. Similarly, whether *delete*(*g*) happens before or after *enter*(*g*, *n*) for any *n*, the outcome is the same.

Some applications can make use of any information returned by a query, no matter how old. Others, such as the orphan detection algorithm, need information that is "recent enough." Also, successive queries by the same client may be executed at different replicas, so it is possible that the second query might return older information than the first one. For some applications, such an anomaly would cause a problem. We allow queries to take timestamps as arguments to capture the notion of "recent enough" and to avoid the anomaly. Timestamps allow a query to identify the updates it depends on.

Using timestamps as query arguments makes queries easy to implement because the timestamp gives a replica a local way of determining whether needed information exists at it: it merely compares the query's timestamp with the its timestamp. When the system is running normally, the needed information, even if very recent, will be at the replica, and the query can be answered without delay and without the replica needing to communicate with other replicas. Only when there are problems such as crashes and partitions will communication with other replicas be needed; these problems are discussed in the next section.

Our method as described so far provides no way for a client to cause updates to be executed in a particular order, but it can be extended to allow the client some control. This is done by having timestamp arguments for updates. If update U is to be after update V, it takes as an argument a timestamp *t* that is greater than or equal to the one returned by V. The service will guarantee that V happens after U and the timestamp returned by V will be greater than *t*. When updates are ordered like this, they no longer need to be commutative; only unordered updates need be commutative.

Ordered updates are useful for some applications. For example, consider a location service that determines the current node of residence for an object. This service might provide a *move* operation that takes the object uid and its new location as arguments. Clearly, if the service is to know where an object is, it must order the moves correctly. Thus we have:

move(ts: timestamp, obj: uid, location: nodeid) returns (timestamp)

Each time *move* is called for object *o*, it is given the timestamp *t* returned by the last call of move for *o* and it returns a timestamp larger than *t*. The timestamps guarantee that the service knows the correct order of moves.

A service with commutative updates is particularly simple to implement. Since the order of commutative operations does not matter, and since all updates are idempotent, a replica can execute updates as soon as it finds out about them, either in processing client requests, or in processing gossip. Ordered updates are slightly harder. One possibility is to delay processing an update if the replica's timestamp is not large enough. However, the delay is really unnecessary since the effects of the update will not be observed until some query runs that is after the update. Therefore the update can return immediately. When a replica performs a query, it must be sure that it knows all needed updates, and that it knows their order. A technique for implementing this is described in [12].

Whether or not there are ordered updates, there is likely to be a need to garbage collect old information such as the information about deleted guardians. We described a general method for doing this garbage collection. Our method makes use of three techniques:

1. Constraints on clients, e.g., that *enter* cannot be called after *delete*.

2. Bounds on how long messages can be in transit. This ensures that very old update messages are discarded.

3. Recording information about the states of other replicas. This information can be used to ensure that gossip messages containing problematical states are discarded.

We showed how these techniques allowed discarding of information in the map service; they can be used similarly in other services.

5 Performance

In this section we compare our technique with other gossip schemes, with voting, and with the work on ISIS [1].

Our method differs from other gossip schemes [5, 20] in two important ways. First, we need only a few replicas, while they require replicas at every client node. Having many replicas is impractical in a system of any size. One consideration is storage of replica state. In some applications, replicas store substantial amounts of information while clients store very little. This dichotomy does not show up in the map service, but it does, for example, in the garbage collector. In that service the replicas store information about inter-object references for the entire system, while individual clients only store information about their own objects that may be referred to by other clients. Another, more important, consideration concerns network traffic. The amount of traffic among replicas grows quadratically with their number, unless the replicas are connected by a broadcast medium. Therefore, we cannot afford a large number of replicas in any environment larger than a single local area net.

The second point is that our method is more powerful than the other schemes. Those schemes allow updates of individual clients to be ordered, and also queries of an individual client are ordered relative to each other and to the client's updates. We allow orderings that involve multiple clients. Queries can use a timestamp argument to require processing after updates of any other client of interest; we saw this in the *refresh* operation. Updates can also use timestamps in this way.

In comparing our method with voting, it is important to realize that voting supports a wider range of applications than our method. We require that updates be idempotent and that those not ordered by clients be commutative; voting does not.

Our method outperforms voting when the system is working well (no crashes or partitions). In this case both reads and writes can take place at a single replica with one message exchange and no locking except what is needed for mutual exclusion at the replica. By contrast, with voting at least some operations (typically writes) must take place at several replicas, must acquire and release locks, and must use more than one exchange of messages with each replica.

To understand the performance of our system in the presence of failures, we look first at partitions and then at crashes. Our method is robust in the presence of partitions and has higher availability than voting. If a partition isolates a client from all replicas, no replication scheme with less than a replica situated at every client node will help. A more common case, however, is a partition that divides nodes into groups containing both replicas and clients.

Our updates can proceed as long as the client's group contains a single replica. A partition could prevent queries from happening if the query required information about an update that was processed on the other side of the partition. There are two cases to consider:

1. A client requires information about an update it did earlier, but the replica that processed the update is separated from the client by a partition that happened after it sent the response to the client but before it communicated with other replicas. This situation is highly unlikely if gossip is frequent. We can make it even more unlikely by sending gossip immediately, in parallel with, or even before, sending the update response to the client.

2. A client requires information about an update done by a different client on the other side of the partition. This situation is impossible if the update was done after the partition formed, because there is no way for the clients to communicate the timestamp of the update. Therefore, the situation is similar to case 1 and can be prevented similarly.

Now we consider the case of crashes. Voting is robust in the presence of crashes, provided only a minority of sites are crashed at a time. Our scheme is also robust, and continues to work even when a majority of sites has crashed.

However, we do have a problem with crashes that is similar to the partition case (1), i.e., the crash happened just after the replica processed an update and responded to the client but before it communicated with other replicas. We can make this unlikely situation even less likely by sending gossip before or in parallel with the update response. If the probability of failure is still considered too high, we can do the following: A replica gossips with another replica (or several other replicas) and waits for an acknowledgment before sending back the user response. Note that this solution differs from voting in that a majority of replicas is still not needed. For example, only two replicas out of five or seven need be involved in an update. Also locking is still not needed. However, the extra communication means that the availability of updates will decline, so the decision here has to be made by trading off update availability versus acceptable probability of being unable to process some queries for some period.

A final point to consider is whether stable storage is needed at the replicas. If replicas process updates without communicating with one another, stable storage seems wise, since otherwise the crash of a single replica could lose some updates forever. However, if replicas gossip (with acknowledgement) before replying, stable storage may not be needed, except to store the replica's part of the timestamp as discussed earlier. The choice here must weigh the availability of updates, the crash independence of the nodes, the cost of stable storage, and the acceptable probability of lost updates.

Now we consider briefly the relationship of our work to ISIS [1]. ISIS supports the same applications that we do, but the technique is different. Instead of using timestamps, ISIS piggybacks information about updates on messages that flow around the system. Thus information about earlier updates will flow on messages sent for later queries. When a query arrives, if the replica does not yet know about some earlier updates, it can retrieve them from the message and process them. In essence, the ISIS technique is similar to our original orphan-detection scheme, in which the entire map had to flow in every message.

The advantage of the ISIS technique is that queries never need to be delayed while a replica communicates with other replicas. By contrast, our method may have an occasional delay. However, our system is more efficient in terms of information that must be remembered and size of messages. In addition, our method generalizes easily to wide area nets, works in the presence of partitions, and avoids a garbage collection problem that exists in ISIS, namely, knowing when it is safe to discard information about old updates.

6 Conclusions

This paper has described a method for constructing highly-available central services for use in a distributed system. The method was discussed with respect to a particular example, the map service. We also discussed how the method could be used for other applications, and compared it with other replication schemes.

As discussed in Section 4, applications can use a service implemented with our method if they obey our constraints. The constraints follow from the fact that we run operations at just one replica. For queries this means that the application must make do with old information; timestamps can be used as query arguments to control the age of the information. Updates must be idempotent and either their order is not determined by clients (as was the case for the map service) or is partially determined by having them take timestamp arguments. Updates not ordered by clients must be commutative; commutativity hides the lack of order, because the meaning of running a group of such updates is the same no matter in what order they run. The service is easier to implement in the commutative case, but can run efficiently in either case.

We have studied several applications that can use our technique. These include garbage collection of objects in a distributed heap, locating movable objects in a distributed system, deletion of unused versions in a hybrid concurrency control scheme, and distributed deadlock detection.

In Section 5 we compared our method with other replication techniques. Our method is better than other gossip schemes for wide area nets and also for cases where the replicas store lots of information that clients need not store. In addition, it provides more power than those methods because updates by different clients can be ordered, and clients can do queries that require information based on updates of other clients. Our method is more limited than voting because of our restriction on operation semantics, but where it can be used it is more efficient both when the system is working well and in the presence of partitions and crashes. Our method supports the same set of applications as ISIS; we suffer an occasional delay in exchange for a drastic reduction in the amount of information flowing in the system.

At present, we are doing an implementation of the map service as part of the Argus system, and we are looking at extensions of the method. Areas under examination are implementing updates that are ordered by clients, combining the method with multi-object transactions, and service reconfiguration.

Acknowledgments

We want to thank Alan Fekete, Sharon Perl, Bill Weihl and the anonymous referees for helpful comments.

References

1. Birman, K., and Joseph, T. Exploiting Virtual Synchrony in Distributed Systems. Proc. of the Eleventh ACM Symposium on Operating Systems Principles, November, 1987, pp. 123-138..

2. Birrell, A., Levin, R., Needham, R., and Schroeder, M., "Grapevine: An Exercise in Distributed Computing". *Comm. of the ACM 25,* 4 (April 1982), 260-274.

3. El Abbadi, A., and Toueg, S. Maintaining Availability in Partitioned Replicated Databases. Proc. of the 5th Conference on Principles of Database Systems, ACM, 1986.

4. El-Abbadi, A., Skeen, D., and Cristian, F. An efficient fault-tolerant protocol for replicated data management. Proc. of the Conference on Principles of Database Systems, ACM, December, 1985.

5. Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. Technical Report MIT/LCS/TR-282, M.I.T. Laboratory for Computer Science, Cambridge, Ma., 1982.

6. Gifford, D.K. Weighted Voting for Replicated Data. Proc. of the Seventh Symposium on Operating Systems Principles, ACM, December, 1979, pp. 150-162.

7. Gray, J. N. Notes on data base operating systems. In *Lecture Notes in Computer Science*, Goos and Hartmanis, Eds., Springer-Verlag, 1978, pp. 393-481.

8. Hwang, D. Constructing Highly-Available Services in a Distributed Environment. S.M. Thesis, M.I.T. Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., December 1987.

9. Lampson, B. W., and Sturgis, H. E. Crash Recovery in a Distributed Data Storage System. Xerox Research Center, Palo Alto, Ca., 1979.

10. Liskov, B., and Scheifler, R. W. "Guardians and actions: linguistic support for robust, distributed programs". *ACM Trans. on Programming Languages and Systems 5,* 3 (July 1983), 381-404.

11. Liskov, B. Overview of the Argus language and system. Programming Methodology Group Memo 40, M.I.T. Laboratory for Computer Science, Cambridge, Ma., February, 1984.

12. Liskov, B., and Ladin, R. Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection. Proc. of the 5th ACM Symposium on Principles of Distributed Computing, ACM, Calgary, Alberta, Canada, August, 1986.

13. Liskov, B., Scheifler, R., Walker, E., and Weihl, W. Orphan Detection. Programming Methodology Group Memo 53, M.I.T. Laboratory for Computer Science, Cambridge, Ma., 1987. Also published in Proc. of the 17th International Symposium on Fault-Tolerant Computing, July 1987.

14. Lundelius, J. Synchronizing Clocks in a Distributed System. Technical Report MIT/LCS/TR 335, M.I.T. Laboratory for Computer Science, Cambridge, Ma., 1984.

15. Marzullo, K. *Loosely-Coupled Distributed Services: A Distributed Time Service.* Ph.D. Th., Stanford University, Stanford, Ca., 1983.

16. Parker, D. S., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., and Kline, C. "Detection of Mutual Inconsistency in Distributed Systems". *IEEE Transactions on Software Engineering SE-9* (May 1983), 240-247.

17. Schlichting, R. D., and Schneider, F. B. "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems". *ACM Trans. on Computing Systems 1,* 3 (1983), 222-238.

18. Walker, E. W. Orphan Detection in the Argus System. Technical Report MIT/LCS/TR 326, M.I.T. Laboratory for Computer Science, Cambridge, Ma., June, 1984.

19. Weihl, W. Distributed Version Management for Read-only Actions. Programming Methodology Group Memo 47, M.I.T Laboratory for Computer Science, Cambridge, Ma., 1986. To appear in IEEE Trans. on Software Engineering, Special Issue on Distributed Systems, December 1986.

20. Wuu, G. T. J., and Bernstein, A. J. Efficient solutions to the replicated log and dictionary problems. Proc. of the Third Annual Symposium on Principles of Distributed Computing, ACM, August, 1984, pp. 233-242.

I. Appendix

In the appendix we prove the correctness of a system implementing a central service; in particular we consider the map service. We begin by setting up a formal model for the operation of the map service. Next we describe in the model the implementation of the service and prove that it preserves certain invariants. Finally we state the specification of the service as seen by the clients, and prove that it is satisfied by the implementation.

I.1 The Model

A system is a connected network of client nodes C_1, ..., C_n and server nodes S_1, ..., S_k. The server nodes represent the central service system, where S_i stands for replica i. Nodes execute by sending messages, receiving messages and updating their local states in response to received messages. We model the execution of a node as a sequence of send and receive events and associate the local updates with the corresponding receive events. The send events occur at the sender node and the receive events at the receiver node.

An execution of system is then a set E of node events and a partial order relation → on E, denoted by $<E, \rightarrow>$. The partial order relation is the smallest relation satisfying two conditions:

1. PO1: Events occurring at the same node are totally ordered.

2. PO2: Let e_1 be a send event and e_2 the corresponding receive event. Then $e_1 \rightarrow e_2$.

The relation → is called *happens before* or *precedes*. Note that it does not imply that messages arrive in FIFO order. The event model is useful in characterizing an unreliable distributed system in which a send event is not necessarily matched by a receive event.

In the following, we assume that crashes cause no loss of server information. We also assume that there is more than one server node.

I.2 Implementation

We now describe the system MP implementing the map service, specify its correct behavior by defining certain invariants, and prove that all executions of the system preserve the invariants.

I.2.1 Execution of the Service

This section describes MP by defining the local data and the events occurring at server nodes. We begin by introducing some auxiliary notation for the precedes relation and define the components of an event. Then we define the local data structures of a server node and the structure of messages. Next, we introduce some useful abbreviations. Finally, we define the operations performed at the server corresponding to the different events.

If event e immediately precedes event e', we denote this by $e \rightarrow_1 e'$. A receive gossip event has two immediate predecessors: the remote send event that supplies the gossip message, denoted $p^r(e)$, and the event that immediately precedes e locally, denoted $p^l(e)$. Send events have one immediate predecessor $p^l(e)$. Receive enter and delete events also have one immediate predecessor $p^l(e)$ because we consider only events at server nodes in the execution of MP. If an event is the first at its node, then $p^l(e)$ is a dummy initial event.

Let e be an event. Node(e) denotes the node at which e occurs and op(e) denotes the type of the event, which may be RecEnter, RecDelete, RecGossip or Send. D(e) denotes the contents of node(e) immediately after e completed. M(e) denotes the data record in the message associated with e. Events have an occurrence time. This is the time of the local clock when the event happened at the server node at which the event occurred. Without loss of generality we assume here that events at a node happen instantaneously.

The local data of a server node (replica) S_i is described by a record D that contains the node's timestamp TS and its part of the timestamp RTS, the set STATE of current associations, the set PEND of pending delete associations, the set OTHERS of multipart timestamps of other server nodes, and the time CLOCK corresponding to some time value read from the local operating system clock:

```
type D = record
        TS:    MultipartTimestamp;
        RTS:   OwnTsPart;
        STATE: SetofAssociations;
        PEND:  SetofDeleteAssociations;
        OTHERS: SetofNodeTsPairs;
        CLOCK: Time;
     end;
```

where an Association is:

```
type Association = record
          UID: NodeName;
          CC: CrashCount;
        end
```

and a DeleteAssociation includes the time of the sender of the delete message and also the timestamp produced by the server that processed the delete.

```
type DeleteAssociation = record
        UID:  NodeName;
        TIME: Time;
        TS:  MultipartTimestamp;
      end;
```

Note that D.CLOCK(e), where e is an event, denotes the value of the CLOCK field at e's node immediately after e completes. This value is set during execution of e by reading a local clock at e's node. Note also that D.CLOCK(e) is the occurrence time of e because of the assumption that events happen instantaneously. Initially, TS and RTS are zero, CLOCK is the initial value of the node's clock, and all other components are empty.

Now we describe the data in the message M of an event. We are concerned here only with messages of events that cause state changes, namely the receive events of updates (enters and deletes) and gossip:

```
type M = record
        NODE:    SendingNodeName;
        DATA:    OpType;
       end
```

where

```
 OpType = Oneof
        RecGossip  (TS: MultipartTimestamp, STATE: SetofAssociations,
             PEND: SetofDeleteAssociations);
        RecEnter    (A: Association, TIME: Time);
        RecDelete  (UID: NodeName, TIME: Time);
      end
```

(A oneof is a discriminated union.) We introduce the following notation: If e is a RecEnter event, then its time is denoted M.TIME(e), its association is M.A(e), and the UID in the association is M.UID(e). Similarly, if op(e) = RecDelete, then its time is M.TIME(e) and its UID is M.UID(e). If op(e) = RecGossip, then the components in the message are M.TS(e), M.STATE(e), and M.PEND(e).

Next we introduce a number of abbreviations. Let ts_1 and ts_2 be multipart timestamps. The merge of two multipart timestamps is defined as

$$\text{merge}\ (ts_1, ts_2) = \min \{\ ts \mid ts \geq ts_1 \text{ and } ts \geq ts_2\ \}.$$

Also for timestamp ts and node n, $ts|_n$ stands for n's part of ts.

Let s be a state of a node and a be an association: The predicate *Isin* defines what it means for a state to reflect an association.

$$\text{a } \mathit{Isin} \text{ s} \equiv$$
$$\exists \text{ b} \in \text{s s.t. a.UID} = \text{b.UID} \wedge \text{a.CC} \leq \text{b.CC}$$

Similarly, let s1 and s2 be two node states. The predicate *Subof* defines what it means for a state to be reflected in another state:

$s1\ \mathit{Subof}\ s2 \equiv$
$\quad \forall\, a\ (\ a\ \mathit{Isin}\ s1 \Rightarrow a\ \mathit{Isin}\ s2\ )$

The binary operation *Combine* defines the state reflecting two other states:

$s1\ \mathit{Combine}\ s2 \equiv$
$\quad \{a \mid a \in s1 \cup s2 \quad \wedge$
$\quad (\ \exists\ b \in s1,\ c \in s2 \text{ s.t. } a.UID = b.UID = c.UID \ \Rightarrow (\ a.CC = \max\ (b.CC,\ c.CC)\ )\ \}$

We express the assumed bound on message delays by a predicate *late*:

$\text{late}(e) \equiv M.TIME(e) + \delta < D.CLOCK(e).$

As discussed in the body of the paper, for the system to work efficiently δ must be large enough to accommodate both the actual delay in sending messages and the clock skew among the client and server nodes.

Now we define the processing associated with the local operations at a server nodes of MP. Processing of any event involves a call to a system routine GetTime to read the local clock. Processing of a send event at a server node corresponds to a message being sent that contains the necessary components of the sender's state. Thus, for event e s.t. op(e) = RecGossip (i.e. $op(p^r(e))$ = Send), M.STATE(e) = $D.STATE(p^r(e))$, M.PEND(e) = $D.PEND(p^r(e))$, and M.TS(e) = $D.TS(p^r(e))$. Only processing of receive enter, delete or gossip events changes the node's state; this processing is defined below. If event e is the first event to be processed by the node, then $D(p^l(e))$ contains the initial values. Also, we assume that if processing of the event does not produce D(e) explicitly, then D(e) has the corresponding values of $D(p^l(e))$.

```
op(e) = RecEnter:
    D.CLOCK(e) = GetTime
    if late(e) then exit
    if ∃ delete association a ∈ D.PEND($p^l(e)$) s.t. a.UID = M.UID(e) then exit
    if M.A(e) Isin D.STATE($p^l(e)$) then exit
    D.RTS(e) = D.RTS( $p^l(e)$) + 1
    D.STATE(e) = D.STATE($p^l(e)$) Combine { M.A(e) }
    $D.TS(e)|_{node(e)}$ = D.RTS(e)

op(e) = RecDelete:
    D.CLOCK(e) = GetTime
    if late(e) then exit
    if ∃ delete association a ∈ D.PEND($p^l(e)$) s.t. a.UID = M.UID(e) then exit
    D.RTS(e) = D.RTS( $p^l(e)$) + 1
    D.STATE(e) = D.STATE($p^l(e)$) — { a | a.UID= M.UID(e) }
    $D.TS(e)|_{node(e)}$ = D.RTS(e)
    D.PEND(e) = D.PEND($p^l(e)$) ∪ { makeDeleteAssociation (M.UID(e), M.TIME(e), D.TS(e)) }

op(e) = RecGossip:
    D.CLOCK(e) = GetTime
```

```
if $M.TS(e) \leq D.TS(e)$ then exit
$D.TS(e) = merge ( D.TS(p^{l}(e)), M.TS(e) )$
$D.STATE(e) = D.STATE(p^{l}(e))$ *Combine* $M.STATE(e)$
            $- \{ a \mid \exists b, b.UID = a.UID, b \in D.PEND(p^{l}(e)) \cup M.PEND(e) \}$
$D.OTHERS(e) = \{ p \mid p \in D.OTHERS(p^{l}(e)) \wedge p.NODE \neq M.NODE(e) \}$
            $\cup \{ makePair (M.NODE(e), M.TS(e)) \}$
$D.PEND(e) = \{ a \mid a \in D.PEND(p^{l}(e)) \cup M.PEND(e)$
                 $\wedge ((a.TIME \geq D.CLOCK(e) - \delta$
                      $\vee \exists p \in D.OTHERS(e)$ s.t. $a.TS \neg\leq p.TS ) \}$
```

I.2.2 Correctness of the Service

In this section we state and prove the invariants maintained by executions of System MP.

Under the assumptions (stated formally below) that server clocks never run backwards and that for any given association clients never perform an update after a delete, the correctness criteria for the implementation of system MP is as follows:

INVARIANT_MP

I1. No Spurious Associations: Let $<E, \rightarrow>$ be an execution of the MP system, e an event in E, and a an association. Then
$a \in D.STATE(e) \Rightarrow$
$\exists$ event $f \in E$ s.t. $op(f) = RecEnter$, $\neg late(f)$, $M.A(f) = a$, and $D.TS(f) \leq D.TS(e)$.

I2. Enters affect the state: Let $<E, \rightarrow>$ be an execution of the MP system and e be an event in E. Then
$op(e) = RecEnter$ and $\neg late(e) \Rightarrow$
$M.A(e)$ *Isin* $D.STATE(e)$
or ($\exists f \in E$ s.t. $op(f) = RecDelete$, $\neg late(f)$,
$M.UID(f) = M.UID(e)$ and $D.TS(f) \leq D.TS(e)$).

I3. Associations remain in the state: Let $<E, \rightarrow>$ be an execution of the MP system and e and f be events in E. Then
$D.TS(e) \leq D.TS(f) \Rightarrow$
$D.STATE(e) - \{ a \mid a \in D.STATE(e)$ and
($\exists$ event h s.t. $op(h) = RecDelete$, $\neg late(h)$,
$M.UID(h) = a.UID$ and $D.TS(h) \leq D.TS(f)$) $\}$
Subof $D.STATE(f)$.

I4. Deletes remove associations: Let $<E, \rightarrow>$ be an execution of the system and e and f be events in E. Then
$op(f) = RecDelete$ and $\neg late(f)$ and $D.TS(f) \leq D.TS(e) \Rightarrow$
$\neg \exists$ association $a \in D.STATE(e)$ s.t. $a.UID = M.UID(f)$.

The invariant is true only if clients obey the constraint:

CLIENT_CONSTRAINT

> Let E be an execution of MP and f and g be events in E s.t. op(f) = RecEnter, op(g) = RecDelete. Then
> $M.UID(f) = M.UID(g) \Rightarrow M.TIME(f) < M.TIME(g)$.

and server clocks obey the constraint:

CLOCK_CONSTRAINT

> Let E be an execution of MP and f and g be events in E s.t. node(f) = node(g).
> Then $f \rightarrow g \Rightarrow D.CLOCK(f) \leq D.CLOCK(g)$.

Note that client clocks must be monotonic in so far as is needed to satisfy the CLIENT_CONSTRAINT.

In much of what follows we will be concerned with enter events whose processing causes an association to be entered into the state. Such events will be referred to as *source* events. We have two categories of such events: *sourceEnter* events and *sourceDelete* events, where

$sourceEnter(e) \equiv (\ op(e) = RecEnter \text{ and } D.RTS(e) > D.RTS(p^l(e))\ ).$
$sourceDelete(e) \equiv (\ op(e) = RecDelete \text{ and } D.RTS(e) > D.RTS(p^l(e))\ ).$

Note that if e is a sourceEnter event, then $\neg$late(e) and $\exists$ association a s.t. $a \notin D.STATE(p^l(e))$ and $a \in D.STATE(e)$ and $a = M.A(e)$. Similarly, if e is a sourceDelete event, then $\neg$late(e) and $\exists$ delete association $a \in D.PEND(e)$ s.t. $a.UID = M.UID(e)$ and $a.TS = D.TS(e)$ and $\neg\ \exists$ delete association $b \in D.PEND(p^l(e))$ s.t. $b.UID = M.UID(e)$.

We now prove that for all executions of system MP, INVARIANT_MP holds. We begin with a very basic Lemma 1 that shows how timestamps relate to the precedes ordering and then in Lemma 2 prove invariant I1.

Lemma 1: Let f and g be events in E s.t. $f \rightarrow g$. Then $D.TS(f) \leq D.TS(g)$.

Proof: Since $f \rightarrow g$, there exists a sequence

$f = h_1 \rightarrow_1 h_2 \rightarrow_1 \cdots \rightarrow_1 h_k = g.$

By inspection of the code, we know that $\forall$ events e_1 and e_2 s.t. $e_1 \rightarrow_1 e_2$, $D.TS(e_1) \leq D.TS(e_2)$. Therefore $D.TS(f) \leq D.TS(g)$. □

Lemma 2: Let e be an event $\in$ E and a be an association in D.STATE(e). Then $\exists$ enter event $f \in E$ s.t. sourceEnter(f) and M.A(f) = a and $D.TS(f) \leq D.TS(e)$.

Proof: The proof is by induction on the size s of the set of predecessors of e. The basis is s = 0. In this case, e is the desired event f. Assume the claim holds for events where size s < n and consider e s.t. s = n. If op(e) $\neq$ RecEnter and op(e) $\neq$ RecGossip, $a \in D.STATE(p^l(e))$ so f exists by the induction assumption and has a proper timestamp by Lemma 1. If op(e) = RecGossip, either $a \in D.STATE(p^l(e))$ or $a \in M.STATE(e)$. In either case the desired event f exists by the induction assumption and Lemma 1.

If op(e) = RecEnter, there are two cases. If processing of e added a to the state, then e is the desired event f. Otherwise, a $\in$ D.STATE(p^l(e)) and f exists by the induction assumption and Lemma 1. □

Next, in Lemma 3 we establish a basic fact about delete associations, namely, that delete associations are added to PEND only by the processing of delete events. Then Lemma 4 proves the I2 part of the invariant: we show that an enter event causes an association to be reflected into the state unless an earlier delete event for its UID has already been processed.

Lemma 3: Let e be an event in E and b a delete association in D.PEND(e). Then $\exists$ f $\in$ E s.t. sourceDelete(f), M.UID(f) = b.UID, D.TS(f) = b.TS, and D.TS(f) $\leq$ D.TS(e).

Proof: By induction on the size of the set of predecessors of e, similar to the proof of Lemma 2. □

Lemma 4: Let e be an event in E. s.t. op(e) = RecEnter and $\neg$late(e). Then M.A(e) *Isin* D.STATE(e) or $\exists$ f $\in$ E s.t. sourceDelete(f) and M.UID(f) = M.UID(e) and D.TS(f) $\leq$ D.TS(e).

Proof: Since e is not late, by the code we see that its processing will ensure that a is reflected into D.STATE(e) unless $\exists$ delete association b $\in$ D.PEND(p^l(e)) s.t. b.UID = a.UID. If a is in the state, we are done, so assume it is not. Then by Lemma 3 the required event f exists. □

Before going on to invariant I3 we introduce a handy abbreviation that characterizes removing of associations and state two useful facts that follow directly from inspection of the processing code.

During the execution E, sets of associations (PEND or STATE) are carried in messages associated with events and are merged into corresponding states at the node where the receive events occur. Thus w.r.t. the partial order defined on E, associations from predecessor states are combined and reflected in successor states. Now, let e and f be events in E s.t. f $\rightarrow_1$ e. If a is an association reflected in D.STATE(f) and not reflected in D.STATE(e), we say that that association a *has been removed from* the state of f by event e. Similarly, if b $\in$ D.PEND(f) and b $\notin$ D.PEND(e), then we say that b *has been removed from* D.PEND(f) by event e.

The following two facts follow directly from inspection of the code.

Proposition 5: Let f and e be events in E s.t. f $\rightarrow_1$ e and b be a delete association $\in$ D.PEND(f). Then either b $\in$ D.PEND(e) or $\forall$ p server nodes, D.OTHERS(e)(p) $\geq$ b.TS. (Here we are treating D.OTHERS as a map from nodes to timestamps.)

Proposition 6: Let f and e be events in E s.t. f $\rightarrow_1$ e and let a be an association $\in$ D.STATE(f). If f = p^l(e), either a *Isin* D.STATE(e) or a is removed from D.STATE(f) by e. If f = p^r(e), then either a *Isin* D.STATE(e) or a is removed from M.STATE(e) by e.

We now turn to invariant I3. We need to prove that for a given execution, if an association in some event's state is missing in some successor event's state, then this successor event is also preceded by a delete event for the missing association. We begin in Lemma 7 by showing that a source event precedes all the events with timestamps larger then its timestamp. In Lemma 8 we show that if an association in some event's state is missing from a successor event's state, then there exists an event preceding the successor event that removes the association. Finally, Lemma 9 proves I3.

Lemma 7: Let f and g be events s.t. sourceEnter(f) or sourceDelete(f) and $f \neq g$. Then $D.TS(f) \leq D.TS(g) \Rightarrow f \rightarrow g$.

Proof: $D.TS(f) \leq D.TS(g) \Rightarrow D.RTS(f) \leq D.TS(g)|_{node(f)}$. Note that for any given value of timestamp ts only the server p could set the value of $ts|_p$ and this is done as part of the processing of some source event at p. Thus, if D.TS(g) has a value t in its component $D.TS(g)|_{node(f)}$, ∃ source event e s.t. D.RTS(e) = t and node(e) = node(f). If g = e we are done. Otherwise ∃ a sequence of ordered events

$$e = h_1 \rightarrow_1 \cdots \rightarrow_1 h_k = g$$

s.t. $t = D.TS(h_i)|_{node(f)}$, i = 1, ..., k, and therefore f → g. □

Lemma 8: Let f and e be events of E s.t. f → e and op(f) = RecEnter, and let a = M.A(f). Assume a ***Isin*** D.STATE(f) and a ¬***Isin*** D.STATE(e). Then ∃ event g, $D.TS(g) \leq D.TS(e)$, s.t. the processing of g removed association a from $D.STATE(p^l(g))$ or M.STATE(g).

Proof: f → e implies there is an ordered sequence

$$f = h_1 \rightarrow_1 \cdots \rightarrow_1 h_k = e.$$

Since a ***Isin*** D.STATE(f) and a ¬***Isin*** D.STATE(e), ∃ $g = h_i$, where $1 < i \leq k$, s.t. a ***Isin*** $D.STATE(h_{i-1})$ and a ¬***Isin*** D.STATE(g). Since a ***Isin*** $D.STATE(h_{i-1})$, this means ∃ association b ∈ $D.STATE(h_{i-1})$) s.t. a.UID = b.UID. By Proposition 6, b is removed from $D.STATE(h_{i-1})$ by the processing of g and $D.TS(g) \leq D.TS(e)$ by Lemma 1. Therefore g is the required event. □

Lemma 9: Let e and g be events in E s.t. $D.TS(e) \leq D.TS(g)$. Let a be an association s.t. a ***Isin*** D.STATE(e) and a ¬***Isin*** D.STATE(g). Then ∃ event f s.t. $D.TS(f) \leq D.TS(g)$, sourceDelete(f), and M.UID(f) = a.UID.

Proof: Without loss of generality we can assume that a ∈ D.STATE(e). Lemma 2 then implies that ∃ h s.t. sourceEnter(h), M.A(h) = a, $D.TS(h) \leq D.TS(e)$, and a ∈ D.STATE(h). Since, a ¬***Isin*** D.STATE(g), $h \neq g$. Since $D.TS(h) \leq D.TS(e) \leq D.TS(g)$, by Lemma 7 h → g. Since a ∉ D.STATE(g), by Lemma 8 ∃ event h', $D.TS(h') \leq D.TS(g)$, s.t. the processing of h' removed a from either $D.STATE(p^l(h'))$ or M.STATE(h'). Consider the event h' and the two possibilities for the removal of a. First, assume a was removed from $D.STATE(p^l(h'))$. This implies by inspection of the code that either op(h') = RecDelete, M.UID(h')=a.UID, and h' is the requested event f, or op(h') = RecGossip and ∃ delete association b ∈

M.PEND(h') s.t. b.UID = a.UID, which by Lemma 3 implies the desired event f exists. Secondly, suppose a was removed from M.STATE(h'). The removal implies $\exists$ delete association b $\in$ D.PEND(p^l(h')) s.t. b.UID = a.UID. As in the first case, by Lemma 3 the existence of the required event f follows. □

Finally, we prove invariant I4 by showing that states associated with events following delete events cannot contain the deleted associations. Lemma 10 proves that delete associations "reach" a node when its timestamp first exceeds that of the corresponding source delete event, Lemma 11 proves that each node "gets to see" a delete association before it is removed, and Lemma 12 proves I4.

> **Lemma 10:** Let f be an event in E s.t. f is a sourceDelete event, and let b be a delete association s.t. b $\in$ D.PEND(f), b.UID = M.UID(f), and b.TS = D.TS(f). Let e be the first event at its node s.t. D.TS(e) $\geq$ D.TS(f). Then b $\in$ D.PEND(e) or the processing of e removed b from M.PEND(e).

Proof: If e = f, the claim follows immediately. If e $\neq$ f, then f $\rightarrow$ e by Lemma 7, so there is an ordered sequence

$$f = h_1 \rightarrow_1 \ \dots \ \rightarrow_1 h_k = e.$$

Since e is the first event at its node with timestamp $\geq$ D.TS(f), node(h_i) $\neq$ node(e) for i = 1, ..., k - 1. By the assumption that b $\in$ D.PEND(f) and Proposition 5 either b $\in$ D.PEND(e) or $\exists$ h_i, $1 < i \leq k$, s.t. b $\notin$ D.PEND(h_i) and $\forall$ nodes p, D.OTHERS(h_i)(p) $\geq$b.TS and in particular, D.OTHERS(h_i)(node(e)) $\geq$ b.TS. If i < k, there must exist gossip event e', node(e) = node(e'), s.t. e' $\rightarrow$ h_i and D.OTHERS(h_i)(node(e)) = D.TS(e') $\geq$ b.TS. But e' $\rightarrow$ h_i means that e' $\rightarrow$ e, which is a contradiction since we assumed that e is the first event at its node with timestamp $\geq$ b.TS. Therefore b $\in$ D.PEND(h_i) for i = 1, ..., k - 1, and either b $\in$ D.PEND(e) or processing of e removes b from M.PEND(e). (Note that the case where processing of e removes b can occur only in a system with exactly two server nodes.) □

> **Lemma 11:** Let f and e be events in E s.t. f $\rightarrow$ e and sourceDelete(f), and let b be the delete association added by f to D.PEND(f) and b $\notin$ D.PEND(e). Then $\exists$ event g, g = e or g $\rightarrow$ e, s.t. node(g) = node(e) and the processing of g removed b from D.PEND(p^l(g)) or M.PEND(g).

Proof: Let e' be the first event at node(e) s.t. D.TS(e') $\geq$ D.TS(f). Then e' = e or e' $\rightarrow$ e. We know by Lemma 10 that b $\in$ D.PEND(e') or processing of e' removed b from M.PEND(e'). If processing of e' removed b from M.PEND(e'), then e' is the desired event g. So assume b $\in$ D.PEND(e'). Since b $\notin$ D.PEND(e), e $\neq$ e' and therefore there is a sequence

$$e' = h_1 \rightarrow_1 \dots \rightarrow_1 h_k = e$$

s.t. node(h_i) = node(e) for i = 1, ..., k. The desired g is the first h_i s.t. b $\in$ D.PEND(h_{i-1}) and b $\notin$ D.PEND(h_i) and clearly g = e or g $\rightarrow$ e. □

> **Lemma 12:** Let g and f be events in E s.t. sourceDelete(f) and D.TS(f) $\leq$ D.TS(g). Then $\neg$ $\exists$ association a $\in$ D.STATE(g) s.t a.UID = M.UID(f).

Proof: If $f = g$ then the claim holds immediately by inspection of the code, so assume $f \neq g$. By **Lemma 7**, $f \rightarrow g$. Now we prove that $\neg \exists$ association $a \in$ D.STATE(g) s.t. a.UID = M.UID(f). The proof is by induction on the size s of the set of predecessors of g in E. The basis is $s = 1$. In this case f is the only predecessor of g, and by inspection of the code we see that processing of g cannot add a to D.STATE(g). Assuming the claim holds for all events where $s < n$, consider event g s.t. $s = n$. If $\exists$ b $\in$ D.PEND(g) s.t. b.UID = a.UID then a is not in D.STATE(g), so assume there is no such b in D.PEND(g). Then by **Lemma 11** $\exists$ event h s.t. $h = g$ or $h \rightarrow g$ and node(h) = node(g) and processing of h removed b from D.PEND(p^l(h)) or from M.PEND(h). If $h = g$, we are done, so consider $h \rightarrow g$. When h was processed, the following were true:

$$\text{b.TIME} + \delta < \text{D.CLOCK(h), and}$$
$$\forall \text{ nodes p (D.TS(f)} \leq \text{D.OTHERS(h)(p))}$$

Since node(h) = node(g) and $h \rightarrow g$, there is a sequence

$$h = h_1 \rightarrow_1 \ldots \rightarrow_1 h_k = g$$

s.t. node(h_i) = node(h), i = 1, ..., k. By the induction assumption, a $\notin$ D.STATE(h_i) for i = 1, ..., k - 1. Therefore, if a is in D.STATE(g), it must have been added by processing of g. If op(g) = RecEnter and M.UID(g) = a.UID, g will have no effect because D.CLOCK(g) > D.CLOCK(h) and therefore the message is late. (Note that here we rely on clients to obey the CLIENT_CONSTRAINT and on server clocks to obey the CLOCK_CONSTRAINT.) If op(g) = RecGossip, there are two possibilities depending on whether the send of the gossip message, i.e., event p^r(g), happens before or after the event h' s.t. node(h') = node(p^r(g)) and D.TS(h') = D.OTHERS(h)(node(h')). If p^r(g) happens after h' then M.TS(g) $\geq$ D.TS(f) and a $\notin$ M.STATE(g) by the induction assumption. Otherwise, processing of g will have no effect because M.TS(g) $\leq$ D.OTHERS(h)(M.NODE(g)) $\leq$ D.TS(h) $\leq$ D.TS(g). □

Theorem 13: INVARIANT_MP holds for all executions of system MP.

I.3 Client Interface

Now we can talk about the correctness of the service from the point of view of the client. The client receives timestamps as results of updates, but is able to observe the state only by executing the refresh queries. Therefore, the correctness condition is expressed in terms of what can be observed by queries.

For a query operation q, let q.TS be the argument timestamp, and q.STATE and q.NEWTS be the state and timestamp returned to the client. For enter operation u, let u.A be the association sent to the service, u.UID be the UID in the association, and u.TS be the timestamp returned to the client in the case where u returns. Similarly, for delete operation u, let u.UID be the UID sent to the service, and u.TS be the returned timestamp in the case where u returns. For enter and delete operations u we use the notation u.TIME to denote the time of the local clock at the client node making the call when u is called

The client specification is:

SPEC _ CLIENT

1. Let u be a delete operation that returns timestamp u.TS . Then for any query q s.t. q.NEWTS $\geq$ u.TS, $\neg$ $\exists$ association a $\in$ q.STATE and a.UID = u.UID.

2. Let u be an update operation that returns timestamp u.TS. Then for any query q s.t. q.NEWTS $\geq$ u.TS, u.A *Isin* q.STATE or $\exists$ a delete operation u' s.t. u.UID = u'.UID.

3. Let q be a query operation that returns. Then q.TS $\leq$ q.NEWTS. Furthermore, if a is an association $\in$ q.STATE, then $\exists$ enter operation u s.t. u.A = a.

We rely on clients to satisfy the client constraint restated here in terms of the client interface:

CLIENT _ CONSTRAINT _ EXTERNAL

Let u be a delete operation. Then for all enter operations u'
u'.UID = u.UID $\Rightarrow$ u'.TIME $\leq$ u.TIME

Before we can show that this specification is satisfied, we need to discuss how servers process query messages, and also how calls are processed at the client. We assume a simplified model of server operation in which delivery of a query message is delayed until the timestamp of the message (i.e., the argument timestamp) is less than or equal to the timestamp of the server. To process the query, the server returns its current state and timestamp.

When a client calls an update operation, this can result in zero or more update events occurring in the service. Note that we do not distinguish here between different calls of update operations with the same arguments. For every call of an enter operation u,

Events(u) = { e | e $\in$ E and op(e) = RecEnter and $\neg$late(e) and e.A = u.A }.

Similarly, for delete operation u,

Events(u) = { e | e $\in$ E and op(e) = RecDelete and $\neg$late(e) and e.UID = u.UID }.

We make the following assumptions about correct client interface behavior:

A1. $\forall$ events e $\in$ E s.t. op(e) = RecEnter or op(e) = RecDelete and $\neg$late(e),
$\exists$ an update u s.t. e $\in$ Events(u).

A2. $\forall$ update operations u that return, $\exists$ e $\in$ Events(u) s.t. e.TS $\leq$ u.TS.

A3. $\forall$ queries q that return, $\exists$ send event e $\in$ E s.t. D.STATE(e) = q.STATE and D.TS(e) = q.NEWTS.

We also assume that server clocks satisfy the CLOCK _ CONSTRAINT. Note that the CLIENT _ CONSTRAINT follows from CLIENT _ CONSTRAINT _ EXTERNAL and assumption A1.

The first case of the client specification is proved as follows. By assumption A2, associated with the delete u there is a delete event e s.t. M.UID(e) = u.UID and D.TS(e) $\leq$ u.TS and $\neg$late(e). By invariant I4, $\exists$ no association a s.t. a.UID = M.UID(e) and a $\in$ D.STATE(e). By I4 we also know that any state with a larger timestamp than D.TS(e) will not contain a. By assumption A3, $\exists$ no such association in the state q.STATE returned by the query.

The second case of client specification is similar. By assumption A2, associated with the enter u there is an enter event f s.t. f $\in$ Events(u) and D.TS(f) $\leq$ u.TS and $\neg$late(f). By invariants I2 and I3 and assumption A1, either u.A *Isin* q.STATE or it was deleted by delete event e $\in$ Events(u') where u' is a delete operation s.t. u'.UID = u.UID.

The third part of the client specification is proved as follows. By A3 we know there is an event e corresponding to the query q s.t. D.TS(e) = q.NEWTS and D.STATE(e) = q.STATE. Since a query is not processed until the replica's timestamp is large enough, we know that D.TS(e) $\geq$ q.TS. Now let a *Isin* q.STATE. Then a *Isin* D.STATE(e). This means there is a b $\in$ D.STATE(e) s.t. b.UID = a.UID and b.CC $\geq$ a.CC. Then by invariant I1 and assumption A1, there exists an update operation u s.t. u.A = b.

This completes the proof that the client specification SPEC_CLIENT is satisfied by the system MP, assuming CLIENT_CONSTRAINT_EXTERNAL, CLOCK_CONSTRAINT, and assumptions A1 to A3.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy

END

DATE

FILMED

6-1988

DTIC